基于 Forth 虚拟机的嵌入式多任务操作系统体系架构研究 *

代红兵，周永录†，安红萍，梅　浩

(云南大学 信息学院，云南省高校数字媒体技术重点实验室，昆明 650223)

摘　要：面对越来越复杂的嵌入式应用需求以及当今嵌入式操作系统研究领域亟待解决的重构、移植、维护、可信、多核、众核等诸多难题，采用 Forth 虚拟机技术，对基于 Forth 虚拟机架构的嵌入式操作系统关键技术进行探索，提出一种具有良好扩展和移植特性、高效精简的基于 Forth 虚拟机的嵌入式多任务操作系统体系架构。该架构采用分类存储映射、Forth 向量定义和用户变量分离，实现了代码共享和多任务管理。实验结果表明，基于 Forth 虚拟机架构的嵌入式操作系统在发挥 Forth 系统固有特性的同时，减少了资源占用，提高了系统的灵活性及运行效率。

关键词：Forth 虚拟机；多任务；嵌入式环境

中图分类号：TP316　　doi: 10.3969/j.issn.1001-3695.2017.08.0863

Research on embedded multitask operating system architecture based on Forth virtual machine

Dai Hongbing, Zhou Yonglu†, An Hongping, Mei Hao

(*School of Information Science & Engineering Yunnan University*, *Digital Media Technology Key Laboratory of Universities in Kunming* 650223, *China*)

Abstract: Faced with more and more complex embedded application requirements and urgent problems of the current embedded operating system research areas, such as reconstruction, transplantation, maintenance, trusted, multi-core, many-core and so on, this paper proposed an embedded multitask operating system architecture, which presents better scalability, transplantation, streamlined and efficient by exploring the key technologies of embedded operating system based on Forth virtual machine technology. The architecture implemented code sharing and multitask managing, by using the methods of classification storage mapping, Forth vector definition and user variable separation. The experiment results demonstrate that the embedded operating system architecture based on Forth virtual machine takes advantage of the inherent characteristics of the system, reduces the resources consumption, and improves system flexibility and operational efficiency.

Key Words: forth virtual machine; multitask; embedded environment

0　引言

在以往嵌入式系统研究过程中，不断碰到现场不间断运行维护的困扰。许多在线运行的嵌入式固件出现 Bug 或要升级时，往往都需要宕机后在现场进行更新与维护，甚至需要撤离现场带入实验室才能解决，将更新、维护好的固件重新烧录到系统中。在某些特殊的应用场合，就需要一种全新的可重构、可扩展并且能够在线交互的实时多任务操作系统。例如在不间断的远程空间观测应用中，就期望有这类操作系统的支持：可创建观测、控制、采集、处理、通信、监测等一系列并发任务，当某一任务的应用程序出现故障时，在系统不宕机和不影响其他任务执行的情况下，通过远程终端强制阻塞该任务，完成在线修改或下载更新后，重新让该任务进入就绪状态，启动运行。进一步考虑，若以上系统是一个解释/编译的双状态系统，那么代码维护将变得更加直接而且迅捷；若应用程序是积木式的架构，那么代码修复只需要简单替换存在 Bug 的模块，而不需要重新加载整个程序。现时是否有满足以上嵌入式领域资源条件有限情况下复杂应用需求的操作系统呢？答案之一便是基于 Forth 虚拟机架构的嵌入式多任务操作系统（FVMOS，embedded multi-task operating system based on the Forth virtual machine architecture）。

Forth 语言本身就是一种过程控制语言和一种快速开发环境，而不仅仅是一个单纯的编程工具，具有很强的交互性、构造性、移植性和自扩展能力，以及高效的生成代码，甚至可以快速构造出一个实时多任务操作系统[1]。Forth 语言自发明以来，先后形成了 FIG-Forth 、Forth-79、Forth-83、ANSI X3.215-1994[2]、

基金项目：国家自然科学基金资助项目（61640205）

作者简介：代红兵（1963-），男，江西赣州人，正高级工程师，硕士，主要研究方向为嵌入式系统、数字电视技术工作；周永录（1965-），男（通信作者），高级工程师，主要研究方向为嵌入式系统（zhylu@126.com）；安红萍（1981-），女，助理研究员，主要研究方向为嵌入式系统；梅浩（1995-），男，硕士研究生，主要研究方向为嵌入式系统.

ISO/IEC 15145 :1997、FORTH-2012[3]等标准。目前，Forth 语言已越来越广泛地用于设计嵌入式软件和固件，例如 FlashForth（PIC18Fxxxx）；PicForth（PIC16F8xx）；hForth（x86 StrongARM）；Quartus Forth（Palm Pilot）、F68KANS（68K）；Forth for the TMS320C50 [4]、Mecrisp-Stellaris（ARM-Cortex）、PunyForth（ESP8266）、AmForth（Arduino AVR8）等等系统，几乎涵盖了所有主流的嵌入式处理器。严格讲，以上这些都仅仅只是 Forth 语言编程环境，真正将 Forth 推向操作系统领域的是美国 Kitt Peak 天文台开发的 Stand Alone Forth11[5]，并一直成为 Forth 操作系统保持至今的标杆。

Stand Alone Forth11 系统，其主体实际是一个具有快速 I/O 能力的多任务操作系统，其最显著的特点是利用了 PDP-11 小型机的页地址映射实现了 Forth 字典的共享。除 ROM 内监控导引程序外，整个系统代码仅为 40K。此后出现的 Stand Alone Forth88[6]，由于缺少页地址映射机制，没有实现代码的复用，每个任务都是一个独立运行的虚拟机，占据 64K 内存。Chuck Moore 在 SEAforth 系列单芯 40 核处理器、100 核处理器[7~9]研究的基础上，推出了第一个精简的嵌入式 Forth 操作系统—Colorforth，其内嵌一个多任务操作系统和基本 I/O 驱动，核心可重入代码仅 2K，现已从最初的 RISC 处理器拓展到多个平台。Colorforth 的测试结果表明，对同一个应用，Forth 的代码量仅有 C 语言的 1%，而且 K 级的代码可以完成传统 M 级操作系统相同的操作[10]。在业界真正获得较大影响的是 Forth 公司推出的 SwiftX，其也内嵌了一个高效精简的 Forth 多任务操作系统 SwiftOS[11]，其特点除了具备良好的移植特性外，就是系统可重构。典型的例子就是如果不考虑终端任务，生成的目标系统中甚至可以不包含文本解释器。

与传统的操作系统架构不同，FVMOS 并不直接运行在硬件处理器上，而是与用户任务程序一样共同运行在 Forth 虚拟机上。因此，FVMOS 的体系架构实质就是多任务的布局，与多任务管理密不可分。虽然在前期研究工作中，依据 Forth 思维，初步提出了一个 EFOS（Embedded Forth Operating System）系统框架[12]，但没有充分体现出基于 FVM 的设计思想，遗留了若干有待深入研究的问题。本文在主流 FVMOS 多任务操作系统体系架构研究的基础上，以可重构、可扩展、可移植、可交互的多任务组织管理为目标，提出一种满足复杂嵌入式应用需求的 FVMOS 体系架构以及相应的多任务管理算法。

1 FVMOS 总体架构

1.1 Forth 虚拟机

FVMOS 的硬件平台既可是硬件的 Forth 堆栈处理器，也可是寄存器型处理器。由于操作系统是在 Forth 虚拟机上运行，这就决定了其实现存在着较大的独特性。

从 Forth 系统结构看，Forth 是一个词典式结构系统。词典由若干字典组成，字典由 Forth 定义（包含 NF 名字场、LF 链接场、CF 代码场、PF 参数场）组成，每个 Forth 定义对应系统中的一个具有独立功能的程序。所有 Forth 定义都以统一的数据结构（词典结构）组织链接在一起（LFA 指向前一 Forth 字），Forth 定义之间遵循严格的有序调用关系，即高层 Forth 定义只能调用低层 Forth 定义，Forth 定义之间的层次关系完全隐含在 Forth 定义当中。字典最底层的 Forth 定义为堆栈处理器指令组成的代码或由汇编指令组成的 Code 定义，上层为冒号定义（高级定义）。

从 Forth 运行机制看，Forth 是一个合并了解释程序和编译程序的双状态系统。Forth 虚拟机直接运行堆栈处理器指令或由汇编指令组成的 Code 代码，所不同的是，对寄存器处理器来说，多了一个地址解释程序（或内层解释程序，机器原语）Next 控制 Code 代码的执行。在 Next 之外，还有一个控制整个系统的主控循环 Quit，即解释编译程序。在 Quit 的控制之下，一个新输入/扫描的字符串若在字典中被找到，其 CFA 将被编译进字典（编译态）；或转向 Next 立即执行（执行态）。若在字典中没有找到，就转换为数值，编译进字典（编译态）；或压入堆栈（执行态）。

1.2 FVMOS

在主流 FVMOS 体系架构研究的基础上，本文提出了 FVMOS 总体架构，如图 1 所示。架构在 CPU 之上的 FVM 实际就是 Forth 的地址解释程序 NEXT，其运行时所用的堆栈是当前运行任务在各自用户变量区里的私有堆栈，系统上电时，使用的是 Stask 的堆栈。FVM 作为一级抽象很好的屏蔽了系统的硬件细节，其指令指针存放在各自任务的返回栈里，只需要一个当前用户变量区指针 UP 就能在任务间进行切换，因此没有传统上下文切换的额外开销。

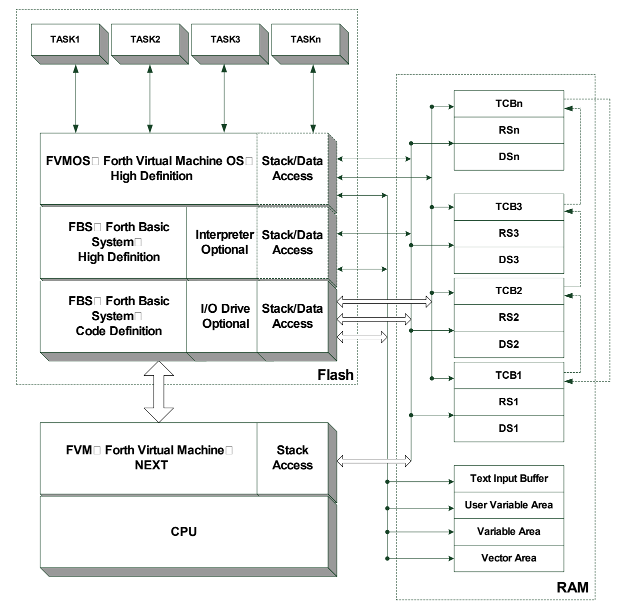


图 1 FVMOS 总体架构

按照以往的布局，需要分别建立 FLASH 和 RAM 两个字典，但这种方式不仅增加了字典管理的开销，而且在实际使用过程中一旦系统掉电，系统就会恢复到上电前的状态，RAM 中动态加载的任务就会丢失。因此本架构将在 FVM 上运行的全

部 Forth 可重入代码存放到 Flash 中。其中，FBS 的底层是 Code 定义，上层是高级定义。在 FBS 之上是向下依赖由高级定义组成的 FVMOS，再往上是由 FVMOS 创建的隶属 Stask 的若干用户任务。为精简目标系统，本架构设计了可裁减的 FBS 选项，若应用系统不考虑在线交互，生成的目标系统中可以不包含文本解释器 Interpreter。

RAM 中有若干与用户任务相对应的由任务控制块 TCB、返回栈 RS、数据栈 DS 等组成的用户变量区，以及文本输入缓冲区 TIB、其他用户变量区和普通变量区等存储项。逻辑上，FBS、FVMOS 都可以进行 RAM 操作（虚框细箭头），但物理上，系统的存取操作被封装在 FBS 的底层 Code 定义中，实际操作是透过这些分类存取指令完成的(粗箭头)。与抢占式 Stand Alone Forth88 不同，协同式 FVMOS 支持终端任务、后台任务和中断任务三种任务类型，虽然三种任务的 TCB 内容有差异，但都连接到多任务循环链表中（右虚线）。

在体系架构方面，FVMOS 架构应当在 Forth 核心词典和主控循环 Quit 之上，这与直接在寄存器处理器上架构操作系统有着明显的差异。此外，其词典式生长结构，也与单体内核结构（模块法）、层次结构以及微内核结构等传统操作系统架构方法有着明显的不同。若仅考虑单核 Forth 虚拟机，其可选的实时多任务操作系统实现方式为：利用 Forth 自身的特点，直接用汇编、C 语言或多或 Forth 自生成器（可异构）生成一个包含 Forth 虚拟机的基本系统 FBS（Forth Basic System），也就是系统任务 Stask（第一个终端任务），之后在其上定义封装：任务管理、内存管理、设备管理、错误陷阱等 FVMOS 原语构件。若存储在 Flash 中的 FBS 和 FVMOS 是可重入的，那么新创建的用户任务 Utask 只在 RAM 用户变量区中建立任务数据空间（运行在同一个 Forth 虚拟机之上，适合于多核 SMP 模式），否则就需要复制 Stask 到新的 Flash 和 RAM 工作区（运行在多个虚拟机之上，适合于多核 AMP 模式或多核离散模型）。同时，初始 Utask 的任务控制块 TCB，并将新的 TCB 插入到系统的 TCB 链表中，进入多任务调度。

在抽象描述方面，与微内核的设计思路不同，由于是在经过一定抽象的堆栈虚拟机上运行，FVMOS 本身就具有较强的跨平台移植性。从层次上看，FVMOS 并没有像传统操作系统那样直接对硬件进行封装，而是运行在虚拟机之上。理论上，排除效率问题，整个 FVMOS 可以不涉及任何汇编语言，其原语和任务体都能从 FBS 的 Forth 定义像搭积木一样构造出来。在异构平台的抽象化描述方面，与 Windows NT、Nanokernel、eCos 等基于硬件抽象层（HAL）设计的操作系统相比，FVMOS 也有其独特的特点和优势。

2　FVMOS 存储架构

2.1　内存分配与管理

在前期研究的基础上，针对嵌入式的存储布局以及 Forth 主流的协同调度算法（Cooperative scheduling algorithm，Round-Robin）的特点，重新优化了设计， 放弃了源自 x86 将 Forth 内存划分为代码段（CS:）、数据段（DS:）、虚拟段（VS:）和堆栈段（SS:）等四个逻辑独立段的设计思路，建立 Flash、RAM、E^2PROM 等三组存取操作定义和相应的指针，分别用于存放代码、数据和参数。

Flash 里存放着可重入的 FBS 和 FVMOS，其分配由 DP 指针决定，存取操作定义包括：,、@I 、IHERE 和 IALLOT 等。RAM 里存放所有程序运行的数据，存取操作定义包括：!、@、HERE 和 ALLOT 等。其中，UP 指针用于指示当前正在运行任务的用户变量区首址（TCB）。一般的嵌入式处理器缺乏页地址映射和内存保护机制，为实现代码保护，可以隐藏 Flash 的显式存取操作。

考虑到 Forth 在线交互过程中允许随时加载新代码的复杂性，在 Forth 系统中，Flash 中的 DP 是严格按地址递增顺序进行分配的。与之对应的 RAM 里的 UP 既可以按实际需求进行简单的顺序分配，也可按动态分配回收算法、缓冲池等技术实现数据区的动态存储管理。

2.2　可重入与用户变量区

可重入代码要求除常量之外不能含有任何私有变量，上述存储管理模式可以支持系统的可重入改造，只要稍作修改，整个 Forth 系统就能实现可重入。Forth 系统的特点决定了系统运行的大多数参数及结果都可以存放在数据栈或返回栈中，要处理的字符串、变量、数组可以保存到系统分配给每个任务的 RAM 数据区中。

引入 Forth 用户变量是实现以上目标的关键。为保证可重入，可以定义一些公共例程去寻址 RAM 数据区里的变量，这些变量就是 Forth 用户变量。Forth 用户变量实际就是地址变量，用户变量定义中 PF 里存储的是该变量在 RAM 中的地址。

用户变量区是 RAM 的一块特殊区域。每个任务都可以共享文本解释器、I/O 驱动等代码，但每个任务都有独自的以用户变量定义的操作数据，这些私有数据存放在 RAM 数据区的不同区域，这个区域就是用户变量区。

2.3　任务控制块

Forth 操作系统多任务调度往往采用的是协同式调度策略，也就是说每个任务的启动时间都是预先确定好的。与其他多任务调度方式不同，FVMOS 的协同式调度是基于虚拟机的，支持终端任务、后台任务和中断任务三种任务类型，在上述内存管理方式下，现场保护仅需要将当前返回栈指针 RP 压入数据栈，并将当前数据栈指针 SP 保存到该任务的用户变量区里。而恢复现场仅需要从该任务的用户变量区里恢复 SP，并将栈顶值存入 RP 指针。FVMOS 的任务控制块（TCB，Task Control Block）就是用户变量区里与多任务调度有关的一块特殊区域，其结构如表 1 所示。

TCB 表中每一项都是通过用户变量 USER 定义的，其中基本表项（前六项）是每个任务的必备项，而附加表项是专门针对终端任务而设置的。附加表项（I/O 驱动）存放的是该终端任

务的 I/O 驱动向量，与该任务具体连接的终端设备有关。附加表项（解释器操作）存放的是该终端任务文本解释器与状态有关的操作向量。需要说明的是 TCB 没有保留返回栈指针，而是将当前返回栈指针放在数据栈栈顶。

表 1　TCB 结构

序号	偏移	名称	描述	类别
1	0	status	任务状态的 xt，UP 指针	
2	2	follower	下一个任务的 TCB 首址	
3	4	rp0	返回栈栈底指针	
4	6	sp0	数据栈栈底指针	基本表项
5	8	sp	数据栈栈顶指针（TOS）	
6	10	CATCHER	错误陷阱	
7	12	BASE	数值转换	
8	14	EMIT	输出字符	
9	16	EMIT?	输出设备就绪	
10	18	TYPE	输出字符串	
11	20	CR	回车	附加表项（I/O 驱动）向量定义
12	22	PAGE	换页	
13	24	AT-XY	移动光标	
14	26	KEY	输入字符	
15	28	KEY?	输入字符就绪	
16	30	ACCEPT	输入字符串	
17	32	DEVICE	设备地址或信息	
18	34	SOURCE	定位输入缓冲区	附加表项（解释器）向量定义
19	36	）IN	输入流中当前偏移量	
20	38	REFILL	填充输入缓冲区	

3　FVMOS 多任务架构及组织管理

FVMOS 的调度原语是 PAUSE，并通过引入可随时置换的 Forth 向量定义 PASS 和 WAKE，不用查找 TCB 表便能实现任务的快速切换。下面重点讨论与操作系统体系架构有关的任务管理算法，针对 Forth 语言和 Forth 系统的特点，本文中算法描述语言采用了 Forth2012 标准。

3.1　Flash 任务定义

在 Flash 的 Forth 字典里创建一个特殊的任务定义，将分配给该任务的用户变量区首址保存到 cfa 代码场里，返回栈、数据栈栈底指针保存到 pfa 参数场里，定义的 cfa 和 pfa 共同组成了任务信息块（TIB，Task Information Block），Task 定义算法如下：

```
/  ds rs us "name" -- ds、rs、us 分别是数据栈、返回栈、追加的用户区大小
: TASK
<BUILDS  / 在 Flash Forth 字典里创建一个任务定义
HERE ，/ 将 HERE（RAM 中 TCB 可用地址）存入 Flash 任务定义的 cfa
( us ) &12 + ALLOT / 在 RAM 中分配用户变量区（前 6 项）
( rs ) ALLOT HERE , / 分配返回栈，将 rp0 保存到 Flash 该任务定义的 pfa[0]
( ds ) ALLOT HERE , / 分配数据栈，将 sp0 保存到 Flash 该任务定义的 pfa[1]
1 ALLOT  / 分隔
DOES> ;   / 执行任务时，将 cfa 压入数据栈
```

入口参数 ds、rs、us 分别是数据栈、返回栈、追加的用户区大小，通过<BUILDS 在 Flash 中创建一个任务头，随后将 RAM 用户变量区中的可用地址 HERE（与之对应，Flash 操作为 IHERE）存入 Flash 任务定义的 cfa（即 TCB 首地），在 RAM 用户变量区中分配 TCB（前 6 项是基本的），分配返回栈，将 rp0 保存到 Flash 该任务定义的 pfa[0]，分配数据栈，将 sp0 保存到 Flash 该任务定义的 pfa[1]。当执行该任务时（DOES>），将 cfa 压入数据栈。

3.2　存储区互操作

位于 Flash 的 TIB 与位于 RAM 的 TCB 之间的互操作由下列定义实现：

```
: TIB>TCB   /  TIB 映射到 tcb [status]
@I ;    /  TIB 映射到 tcb [rp0]
: TIB >RP0
I-CELL+ @I ;
: TIB >SP0  /  TIB 映射到 tcb [sp0]
I-CELL+ I-CELL+ @I ;
: TIB >SIZE  /  TIB 换算用户变量区尺寸 SIZE
DUP TIB >TCB SWAP TIB >SP0 1+ SWAP - ;
```

其中，与 RAM 操作@不同，@I 是从 Flash 里取数。系统保留了 Forth 的习惯，将常规的存取操作符视同为 RAM 操作。

3.3　任务用户变量区初始化

任务创建后，TASK-INIT 负责初始化该任务用户变量区里的 TCB 和堆栈区，设置缺省进制为十进制（Decimal），在尚未链接任务体前，将该任务设置为 PASS 睡眠状态。TASK-INIT 定义算法如下：

```
: TASK-INIT
DUP TIB>TCB OVER TIB >SIZE 0 FILL / 初始化任务用户变量
/ 区里的 TCB 和堆栈区
DUP TIB >SP0 OVER TIB>TCB &6 + ! /  tcb[sp0]=tib[sp0]
DUP TIB >SP0 CELL- OVER TIB>TCB &8 + !  /  tcb[TOS]
DUP TIB >RP0 OVER TIB>TCB &4 + ! /  tcb[rp0]=tib[rp0]
&10 OVER TIB>TCB &12 + !  /  tcb[base]=10
TIB>TCB TASK-SLEEP ;       /  tcb[0]=PASS
```

入口参数为任务头执行时 DOES>弹出的 TIB，通过互操作转换为 TCB 地址，然后对 RAM 用户变量区的 TCB 和堆栈区进行清除，将保存在 Flash 任务头中的 sp0 存储到 tcb[sp0]，计算出栈顶，保存到 tcb[TOS]，将保存在 Flash 任务头中的 rp0 存

储到 tcb[rp0]，设置缺省十进制，最后将初始任务状态设置为 PASS。

多任务存储架构如图 2 所示。

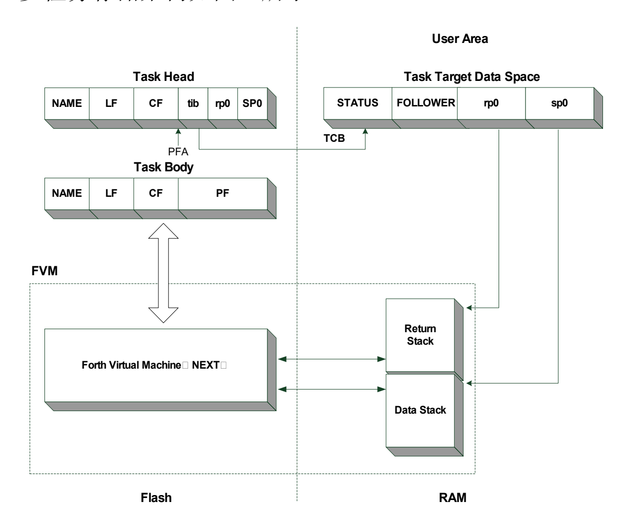


图 2　多任务存储架构

3.4　TCB 循环链表

在完成上述创建、初始化、链接任务体之后，还需要初始化和构建多任务 TCB 循环链表，ADD-FIRSTTASK 初始化第一个终端任务，将其 tcb [status]设置为 WAKE，并将 FOLLOWER 指向自身。ADD-FIRSTTASK 定义算法如下：

```
: ADD-FIRSTTASK
WAKE STATUS !  / 将当前任务的 tcb [status]设置为 WAKE
UP@ FOLLOWER ! ;  / 将 FOLLOWER 指向自身
```

通过 ACTIVATE 原语激活，多任务启动后，允许随时追加新的任务，ADD-NEXTTASK 将 tcb 所指的新任务 TCB 表插入到多任务循环链表中。ADD-NEXTTASK 定义算法如下：

```
/  tcb --
: ADD-NEXTTASK
['] PAUSE DEFER@ >R  / 将 PAUSE 里的 xt（MULTITASK-SCHEDULE
/ 或 NOOP）保存到返回栈
SINGLE  / 停止多任务，PAUSE 被设置为 NOOP
FOLLOWER @  / 得到当前任务的下一个任务 tcb'
OVER
FOLLOWER !  / 将 tcb 作为当前任务的下一个任务
SWAP CELL+
!  / 将 tcb 任务的下一个任务指向当前任务之前的下一个任务
R> IS PAUSE ;  / 恢复多任务
```

入口参数为新添加任务的 TCB 首址。考虑到 TCB 表的访问属临界资源，为支持动态任务添加，一开始就将任务调度向量保存到返回栈上。在关闭多任务调度之后，取出 FOLLOWER 指针（当前任务指向的下一个任务），将新任务的 tcb 作为当前任务的下一个任务，将当前任务的下一个任务作为新任务的下一个任务。完成 TCB 循环链表的直接插入之后，从返回栈恢复任务调度向量，并随即启动任务调度。由于不涉及到优先级调度，因此本算法的 TCB 循环链表维护就变得十分简洁，不需要插入排序操作，算法复杂度为 $O(1)$。过程最后启动了多任务调度，新添加的任务将得到优先执行。

4　实验评估

在 Arduino 嵌入式硬件平台上，借助开源的 Forth 虚拟机，实现了 FVMOS 基本框架和多任务管理算法。实验环境中，在系统上电初始化阶段，先设计了含有一个终端任务 TASK1 和一个后台任务 TASK2 的实验程序如下：

```
: MS ( n -- ) PAUSE 0 ?DO 1MS LOOP ; / 每隔 n 毫秒调用 PAUSE 等待
VARIBLE M  /  TASK1 中 TASK2 使用的全局变量 M
: INITM ( -- ) 0 M ! ;
$40 $40 0 BACKGROUND-TASK TASK2 / 建立后台任务 TASK2，在 FLASH
/ 中创建任务头，分配用户变量区
: TASK2-BODY ( -- ) BEGIN 1 M +! &10 MS AGAIN ; /定义 TASK2 任务体
: STARTTASKER ( -- )  / 初始化并启动多任务
TASK2 BACKGROUND-INIT  / TASK2 初始化，在 RAM 中创建 TCB2
TASK2 TIB>TCB ACTIVATE TASK2-BODY  / 连接 TASK2 任务体
ADD-FIRSTTASK  / 建立 TCB1 循环链表
TASK2 TIB>TCB ADD-NEXTTASK  / 将 TCB2 加入循环链表
MULTI ;  / 启动多任务
: TASK-TURNKEY ( -- )  / 上电启动向量
APPLTURNKEY INITM STARTTASKER ;
```

在系统上电启动运行后，通过终端任务 TASK1，建立新的后台任务 TASK3，通过命令行或文件方式动态载入以下程序，实验代码如下：

```
VARIBLE N  /  TASK1 中 TASK3 使用的全局变量 N
: INITN ( -- ) 0 N ! ;
$40 $40 0 INTERRUPT-TASK TASK3  / 建立中断任务 TASK3，在 FLASH
/ 中创建任务头，分配用户变量区
: TASK3-BODY ( -- ) BEGIN 1 N +! &10 MS AGAIN ; / 定义 TASK3 任务体
: STARTTASK3 ( -- )  / 初始化并启动 TASK3
TASK3 BACKGROUND-INIT  /  TASK3 初始化，在 RAM 中创建 TCB3
TASK3 TIB>TCB ACTIVATE TASK2-BODY  / 连接 TASK3 任务体
INITN
TASK3 TIB>TCB ADD-NEXTTASK ;  / 将 TCB3 加入循环链表
```

在终端任务 TASK1 里定义后台任务 TASK2，多任务启动后，TASK1 在 TASK2 任务体每次调用 MS 时执行一次；TASK2 在 TASK1 每次等待键盘输入时执行（内嵌 PAUSE 向量）。此后，若有动态重构多任务系统的需求，可以在并发运行的终端任务 TASK1 里定义新的后台任务 TASK3，并通过任务初始化 BACKGROUND-INIT、激活连接任务体 ACTIVATE 以及加入循环链表 ADD-NEXTTASK 等操作将 TASK3 加入多任务循环。第一个终端任务 TASK1 实际就是系统任务（Stask），除包含了 FBS 和 FVMOS 两个部分外，还涵盖了系统上电前定义的所有任务以及系统运行后通过各终端任务动态加载的新任务，因此可以通过各终端任务对所有任务进行 SLEEP、WAKE、STOP 等

有效控制。实验表明，系统运行稳定可靠，相关算法能够达到系统可重构、可扩展、可移植、可交互的多任务组织管理目标。

与文献[6]基于 CPU 调度的 Stand Alone Forth88 多任务系统相比，本文提出的 FVMOS 多任务体系架构有诸多的优点。在嵌入式有限存储空间利用方面，基于FVM的架构缩减了TCB的规模。任务切换仅仅只需保留或恢复 SP 指针就能实现，这样就可将 TCB 中数十项 CPU 映像存储压缩到只需 6 个单元。在重构、扩展、移植方面，整个多任务管理系统建构在系统的 FBS 高级定义之上，做到与硬件无关。在系统简洁性方面，特殊的调度方式和精简的 TCB 结构，使得系统不需要新建和维护任务队列。与 Stand Alone Forth88 中创建任务的同一层次代码量（LOC）相比，至少压缩了 10 倍，整体压缩了 16 倍，如表 2 所示。在快捷性方面，除了将常规带优先级的时间片轮转调度算法复杂度从 $O(n^2)$ 降低到 $O(n)$ 之外，所有多任务管理算法的复杂度均为 $O(1)$ 。

表 2　代码量（LOC）对比

对比项目	Stand Alone Forth88	FVMOS
TASK	136	12
MULTITASKER	2500	155

5　结束语

从以上分析可以看出，FVMOS 与传统方式有本质的不同，并不存在严格区分的模块、层次、微内核等传统结构。虽然系统天生具有可重构特性，甚至构件化特征，但与采用构件化操作系统也不尽相同。Forth 特有的堆栈和字典式结构决定了其体系架构和运行机制的特殊性。一方面，Forth 特殊的穿线编码（Threaded Code），使 Forth 堆栈机的词典式结构本身就是一个可动态、交互扩展的开放性程序库。除此之外，Forth 虚拟机运行环境还为系统提供了难得的跨平台能力。但另一方面，也正是因为这些特性，给基于虚拟机的多任务组织管理带来了难题。例如，由于所有过程（包括任务调度）都必须在 Quit 控制之下，因此经典的多任务管理算法难以直接派上用场。虽然实时性还有待提高，但本文研究的 FVMOS 体系架构以及多任务管理算法具有一定的现实指导意义。

参考文献：

[1] 代红兵. 新型、高效微机 Forth 语言的研制 [J]. 中国科学院研究生院学报, 1993, 10 (1): 62-69.

[2] ANSI X3. 215-1994 American National Standards for Information Systems Programming Languages Forth [S]. New York: American National Standards Institute, 1994.

[3] Forth-Standard-Committee. FORTH-2012 [S/OL]. (2015) . https://forth-standard. org/.

[4] FORTH. Inc. Featured Forth Applications [J/OL]. (2009) . http://www. forth. com/resources/app Notes.

[5] Thomas E. McGuire. Kitt Peak Multi-Tasking FORTH-11 [J]. The Journal of Forth Application and Reseach, 1984, 2 (2): 57-67

[6] 代红兵. 高效微机实时多任务操作系统设计与实现 [J]. 中国科学院研究生院学报, 1993, 10 (3): 283-292.

[7] James Rash. Space-related applications of forth [J/OL]. (2006-09) . http://forth. gsfc. nasa. gov/.

[8] Caffrey R T. Forth in space: interfacing SSBUV: a scientific instrument, to the space shuttle [J]. ACM Sigforth Newsletter, 1993, 4 (3): 1-8.

[9] IntellaSys, A TPL Group Enterprise. SEAforth 40C18 Scalable Embedded Array Processor [EB/OL]. (2008) . http://www. intellasys. net/templates/trial/content/SEK _40C18_DataSheet_1. 1. pdf.

[10] Mikiten B C, Mikiten S, Orr J L. A Forth-based real-time in-flight monitoring system [C]// Proc of the 2nd & 3rd Annual Workshops on Forth. New York: ACM Press, 1991: 31-34.

[11] Forth Inc. SwiftX Cross Compilers for Embedded Systems Applications [EB/OL]. (2016) . https://www. forth. com/embedded/.

[12] 杨为民, 代红兵, 安红萍, 等. 一种新的嵌入式 Forth 实时操作系统的研究 [J]. 云南大学学报: 自然科学版, 2013 (S2): 96-103.

[13] Paul Frenger. Forth and AI Revisited: BRAIN. FORTH [J]. ACM SIGPLAN Notices, 2004, 39 (12): 11-16.

[14] Stephen Pelc. Programming Forth [M]. [S. l.] : MicroProcessor Engineering Limited, 2011: 97.

[15] 代红兵, 杨为民, 王丽清, 等. 多目标Forth自生成器的研究与实现 [J]. 计算机应用研究, 2014, 31 (4): 1109-1114.

[16] The Forth Interest Group. Forth Compilers Page [EB/OL]. (2009) . http://www. forth. org/compilers. html.

[17] Frederic P Miller. Colorforth [M]. Alphascript [S. l.] , 2010: 28.

[18] Frenger P. Hard Java [J]. Acm Sigplan Notices, 2008, 43 (5): 5-9.

[19] Hanna D M, Jones B, Lorenz L, et al. An embedded Forth core with floating point and branch prediction [C]// Proc of IEEE International Midwest Symposium on Circuits and Systems. 2013: 1055-1058.